

# Nucleus C++ FILE

Reference Manual



0001152-001 Rev 100

Copyright (c) 2002  
Accelerated Technology, Inc.  
720 Oak Circle Dr. E.  
Mobile, AL 36609  
(251) 661-5770



## Related Documentation

**Nucleus PLUS Reference Manual**, by Accelerated Technology, Inc., describes the operation and usage of the Nucleus PLUS kernel.

**Nucleus PLUS Internals**, by Accelerated Technology, Inc., describes, in considerable detail, the implementation of the Nucleus PLUS kernel.

**Nucleus FILE Reference Manual**, by Accelerated Technology, Inc., describes the operation and usage of Nucleus FILE.

**Nucleus FILE Internals**, by Accelerated Technology, Inc., describes, in considerable detail, the implementation of the Nucleus FILE component.

**Nucleus C++ BASE Reference Manual**, by Accelerated Technology, Inc., describes the operation and usage of the Nucleus C++ BASE component.

**Nucleus C++ PLUS Reference Manual**, by Accelerated Technology, Inc., describes the operation and usage of the Nucleus C++ PLUS component.

## Style and Symbol Conventions

Program listings, program examples, filenames, menu items/buttons and interactive displays are each shown in a special font.

Program listings and program examples - Courier New

Filenames - COURIER NEW, ALL CAPS

Interactive Command Lines - **Courier New, Bold**

Menu Items/Buttons – *Times New Roman Italic*

## Trademarks

MS-DOS is a trademark of Microsoft Corporation

UNIX is a trademark of X/Open

IBM PC is a trademark of International Business Machines, Inc.

## Additional Assistance

For additional assistance, please contact us at the following:

Accelerated Technology, Inc.

720 Oak Circle Drive, East

Mobile, AL 36609

800-468-6853

251-661-5770

251-661-5788 (fax)

[support@atinucleus.com](mailto:support@atinucleus.com)

<http://www.acceleratedtechnology.com>

Copyright (©) 2002, All Rights Reserved.

Document Part Number: 0001152-001 Rev 100

Last Revised: June 8, 2001





# Contents

Chapter 1-Nucleus C++ FILE User's Guide.....	1
About this Manual .....	2
What is in this User's guide? .....	2
What is Nucleus C++ FILE?.....	2
Nucleus C++ FILE Source Files .....	2
Portable Source Files .....	3
Target Dependent Source Files .....	3
Application Dependent Source Files.....	3
Optimizations for Embedded Systems .....	3
File System Initialization and Startup .....	4
Nucleus C++ FILE Classes.....	4
Using Nucleus C++ FILE .....	5
Using the File System 1-2-3 (class NppFILE).....	5
Create and Initialize the NppFILE Object .....	5
Getting A Pointer to the File System Component.....	5
Threads and the File System.....	5
Threads, Default Devices, and Current Working Directories .....	5
Using Storage Devices 1-2-3 (class NuFileDevice).....	6
Create and Initialize the NuFileDevice Object .....	6
Getting A Pointer to a Storage Device.....	6
Accessing Storage Devices .....	6
Accessing Storage Devices Attributes .....	7
Formatting your Device.....	7
Aborting all Pending Operations on a Device .....	8
Using Directories 1-2-3 (class NuDirectory) .....	8
Creating a Directory Object.....	8
Accessing the Path Attributes of a Directory.....	8
Making a Directory on the Device.....	8
Removing a Directory from the Device.....	9
Iterating the Contents of a Directory .....	9
Using Files 1-2-3 (class NuFile).....	9
Creating and Subsequently Opening Files.....	10
Opening a File on the Device .....	10
Closing a File on the Device.....	10



Renaming a File on the Device .....	11
Deleting a File from the Device .....	11
Accessing File Attributes .....	11
Seeking the File Pointer .....	12
Writing to a File .....	12
Reading from a File .....	12
Truncating File Contents .....	12
Flushing the File Buffers .....	12
Nucleus C++ FILE SIMPLE Demonstration Application .....	13
Files and File Locations .....	13
Downloading and Executing the Demo .....	14
What Storage Device is in Use? .....	14
Assertions in Example Code .....	14
Does Nucleus C++ PLUS Exist? .....	15
Startup and Initialization .....	15
Streaming Statistical Output .....	17
The Demonstration System Task .....	20
Create a scratch compare buffer. ....	22
Become a user. ....	23
Open the device for use.....	23
Create the directory.....	23
Create the file.....	24
Seek to the beginning and write.....	24
Seek back to the beginning and read.....	24
Rename the file.....	25
Delete the file.....	25
Go back to the root directory.....	25
Remove the directory.....	25
Close the device.....	26
Release the user.....	26
Chapter 2-Nucleus C++ FILE Class Reference .....	27
class NuFile.....	28
NuFile::NuFile .....	29
NuFile::NuFile .....	30
NuFile::~NuFile .....	32
NuFile::Open.....	33
NuFile::Open.....	36
NuFile::IsOpen.....	40
NuFile::Close .....	41
NuFile::SeekToEnd.....	42
NuFile::SeekToBegin.....	43
NuFile::Seek .....	44
NuFile::Read .....	46
NuFile::Write .....	48
NuFile::Truncate .....	49
NuFile::Flush .....	50
NuFile::Rename .....	51
NuFile::Rename .....	53



NuFile::GetFileName.....	54
NuFile::GetPathName.....	55
NuFile::Delete.....	56
NuFile::Delete.....	58
NuFile::GetDescriptor.....	59
class NuDirectory .....	60
NuDirectory::NuDirectory.....	61
NuDirectory::NuDirectory.....	62
NuDirectory::~~NuDirectory.....	63
NuDirectory::Make .....	64
NuDirectory::Remove.....	65
NuDirectory::GetFirst.....	66
NuDirectory::GetNext.....	68
NuDirectory::Done .....	70
NuDirectory::GetPath .....	71
NuDirectory::SetPath.....	72
class NppFile : public NppComponent .....	73
NppFile::NppFile .....	74
NppFile::Initialize .....	75
NppFile::~~NppFile.....	76
NppFile::BecomeUser.....	77
NppFile::ReleaseUser .....	78
NppFile::GetNumberUsers .....	79
NppFile::GetNumberUserFiles .....	80
NppFile::GetNumberDevices.....	81
NppFile::GetMaxSectors .....	82
NppFile::GetLockMethod.....	83
class NuFileDevice .....	84
NuFileDevice::NuFileDevice.....	85
NuFileDevice::Initialize.....	86
NuFileDevice::~~NuFileDevice .....	87
NuFileDevice::SetFormatParameters.....	88
NuFileDevice::GetFormatParameters .....	89
NuFileDevice::Format .....	90
NuFileDevice::Open .....	92
NuFileDevice::Close.....	93
NuFileDevice::Abort.....	94
NuFileDevice::SetDefaultDevice.....	95
NuFileDevice::GetDefaultDevice.....	96
NuFileDevice::GetCurrentDirectory.....	97
NuFileDevice::SetCurrentDirectory .....	98
NuFileDevice::GetDeviceNumber .....	99
NuFileDevice::GetDevice.....	100
NuFileDevice::PrependDeviceNameToName .....	101





# Nucleus C++ FILE User's Guide

About this Manual

What is C++ FILE?

Nucleus C++ FILE Source Files

Optimizations for Embedded  
Systems

File System Initialization and Startup

Nucleus C++ FILE Classes

Nucleus C++ FILE SIMPLE  
Demonstration System

### About this Manual

This manual is intended to give readers a high level overview of Nucleus C++ FILE so they are better prepared to take advantage of its specific design goals and features.

### What is in this User's guide?

This guide contains explanations of the various source files, what demos are included, application startup, and the various classes included in Nucleus C++ FILE.

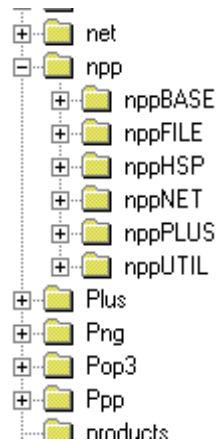
### What is Nucleus C++ FILE?

Nucleus C++ FILE is a C++ class interface into Nucleus FILE, a FAT16/FAT32™ compatible file system specifically designed to work in embedded systems in conjunction with the Nucleus PLUS real-time operating system.

The file system has been encapsulated into the Nucleus C++ component, `NppFILE`. Constructing an `NppFILE` object initializes the file system. All of the details are managed for you. Then, a set of classes is used to create, edit, and manage devices, files, and subdirectories.

### Nucleus C++ FILE Source Files

Nucleus C++ FILE source files are delivered in the `\NPPFILE` sub-directory under the Nucleus C++ `\NPP` directory.



**Nucleus C++ FILE directory is `nppFILE`**



## Portable Source Files

In the `NPP\NPPFILE` sub-directory you will find the following portable files. All Nucleus C++ FILE targets share these files and they are maintained on a product wide basis.

File	Description
NPPFILE.H NPPFILE.CPP NPPFILE.INL	Source files for class <code>NppFILE</code> , <code>NuFile</code> , and <code>NuDirectory</code> .
NPPFILED.H NPPFILED.CPP NPPFILED.INL	Source file for class <code>NuFileDevice</code> .

## Target Dependent Source Files

There aren't any target dependent files in Nucleus C++ FILE. If for some reason a particular version of Nucleus C++ requires special software, the convention is that the files will be named `NPPFILES.H`, `NPPFILES.INL`, and `NPPFILES.CPP` and located in the `npp\nppFILE` sub-directory. Please refer to the target specific notes for your version of Nucleus C++ FILE to check if there are any target specific files.

## Application Dependent Source Files

In the `NPP\NPPBASE` sub-directory you will find an application dependent file, `NPPAPP.H`. Each Nucleus C++ application will have a different version of this file. It is used to specify application specific parameters for all Nucleus C++ components used in that particular application.

File	Description
NPPAPP.H	This file contains all application tuning settings for Nucleus C++. By setting various preprocessor switches, you can include or exclude various features from the build to minimize the footprint of your application.

## Optimizations for Embedded Systems

There aren't any optimization options specific to Nucleus C++ FILE. However, there are many options you can configure within Nucleus FILE. Please refer to the Nucleus FILE documentation for detailed information. The Nucleus C++ FILE component, `NppFILE`, initializes the file system present within your Nucleus FILE library, `FILE.LIB`. Therefore, please make sure Nucleus FILE has been properly configured to match your target hardware prior to building your Nucleus FILE library.



## File System Initialization and Startup

In the Nucleus C++ BASE user's guide you will find a discussion of the Nucleus C++ component architecture. Please read this first for an understanding of the architecture and how startup works in a Nucleus C++ system. Here we will limit our discussion to Nucleus C++ FILE startup and initialization.

The `NppFILE` component is responsible for the initialization of the underlying Nucleus FILE system. It can be created on either the system startup thread or the multitasking startup thread, but the `Initialize` member must be called on a multitasking thread. Below is a typical example.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Nucleus C++ FILE component.
    NppFILE* nppFILE = new NppFILE;
    nppFILE->Initialize();

    // Nucleus C++ FILE device.
    NuFileDevice* nppFILE_Device = new NuFileDevice( 0 );
    nppFILE_Device->Initialize();

    #if (NU_STATIC_OBJECT_SUPPORT)
        // ***NOTICE*** if there are any objects declared at file
        // scope, their constructors will be called now.
        NppSTATIC* nppSTATIC = new NppSTATIC;
        nppSTATIC->Initialize();
    #endif

    ... setup your application here.
}
```

## Nucleus C++ FILE Classes

Nucleus C++ FILE provides classes for the four main parts of Nucleus FILE.

Socket class	Description
<code>NppFILE</code>	This class encapsulates the Nucleus FILE system.
<code>NuFileDevice</code>	Used to initialize and use a storage device.
<code>NuFile</code>	Used to create, edit, and manage files.
<code>NuDirectory</code>	Used to create, edit, and manage directories.



## Using Nucleus C++ FILE

Using the classes in Nucleus C++ FILE is very easy. The following sections discuss creating the file system, storage devices, subdirectories, and files.

### Using the File System 1-2-3 (class NppFILE)

The steps and code required for each step are outlined below.

#### Create and Initialize the NppFILE Object

NppFILE objects can be constructed anywhere you like: local on the stack; global at file scope; or dynamically via a call to operator new. The only constraint on the following section of code is that the `Initialize` member must be called on a multitasking thread.

```
// Create the Nucleus C++ FILE component.
NppFILE* nppFILE = new NppFILE;
nppFILE->Initialize();
```

#### Getting A Pointer to the File System Component

The Nucleus C++ FILE component, NppFILE, has been designed using the singleton design pattern. Therefore, you can get a pointer to the singleton instance of the file system at any time by calling the `Instance` method of the class.

```
// Get a pointer to the file system.
NppFILE* nppFILE = NppFILE::Instance();
```

#### Threads and the File System

Threads must be registered prior to using Nucleus FILE. Registration must occur after the NppFILE component has been initialized. When a thread is completed its file system tasks, it can return its resources by un-registering. This is easily accomplished using the `BecomeUser` and `ReleaseUser` members of the NppFILE component.

```
// Register with the file system.
nppFILE->BecomeUser();

// Use the file system, storage devices, sub-directories, files, etc.
// ...

// Un-Register with the file system.
nppFILE->ReleaseUser();
```

#### Threads, Default Devices, and Current Working Directories

Each thread has a default device. This is the device where the thread's file operations are performed when incomplete path information is supplied. For each thread, Nucleus FILE also maintains a current working directory for each device. This is a directory that incomplete filenames are relative to when a complete path is not specified.



### Using Storage Devices 1-2-3 (class NuFileDevice)

The steps and code required for each step are outlined below. The `NuFileDevice` class is used to manage access to storage devices in Nucleus FILE.

#### Create and Initialize the NuFileDevice Object

Storage devices are identified in Nucleus FILE using a numbering scheme. They are assigned device numbers that are associated with the drive letter for the device. For example, the “A:” drive is device 0; the “B:” drive is 1; the “C:” drive is 2; and so on.

To create a storage device object, simply construct a `NuFileDevice` object and call its `Initialize` member. The constructor takes an integer argument corresponding to the device number. Storage device objects must be created and initialized prior to using any storage device services.

```
// Create and initialize a Nucleus C++ FILE device.
NuFileDevice* nppFILE_Device = new NuFileDevice( 0 /*"A:"*/ );
nppFILE_Device->Initialize();
```

#### Getting A Pointer to a Storage Device

After devices are created, you can access them from the singleton file system using the `GetDevice` class member of class `NuFileDevice`.

```
// Get a pointer to the "A:" device.
NuFileDevice* device = NuFileDevice::GetDevice( 0 /*"A:"*/ );
```

#### Accessing Storage Devices

A thread that uses a storage device within Nucleus FILE must open the device using the `Open` member prior to executing any device operations. When a thread has completed its file system tasks, it can return resources by calling the `Close` member.

```
// Open the storage device.
device->Open();

// Access the storage device, create files, sub-directories, etc.
// ...

// Close the storage device to return resources.
device->Close();
```



## Accessing Storage Devices Attributes

Since each thread has a current working directory for each device in Nucleus FILE, file operations are relative from each working directory on each particular device. For example, if `\My_Current_Working_Directory` is the current working directory for device 0, all files with incomplete path information, targeted for the "A:" drive, will be located in `A:\My_Current_Working_Directory`. Therefore, `A:My_filename.text` has an absolute name `A:\My_Current_Working_Directory\My_filename.text`.

```
// Retrieve the current working directory for this thread.
CHAR path[EMAXPATH];
device->GetCurrentDirectory( path );

// Change the current working directory.
strcpy( path, "\\Another_Current_Working_Directory" );
device->SetCurrentDirectory( path );
```

Similarly, when a thread has a default device set, all file operations are relative from that device. For example, setting the device associated with device number 0 ("A:") as the default device will cause all files with incomplete path information to be located on that device. For example, if we open `My_filename.text` with a current working directory of `\My_Current_Working_Directory`, the absolute name of the file will be `A:\My_Current_Working_Directory\My_filename.text`.

```
// Make this the default device.
device->SetDefaultDevice();

// Get the default device. Note: this is a static class member.
device = NuFileDevice::GetDefaultDevice();
```

## Formatting your Device

You can access the format parameters on the device by calling the `GetFormatParameters` and `SetFormatParameters` members. The following example accesses the current format parameters from a device that already has them set within Nucleus FILE (like an ATA storage device). Some devices require you to provide format specifications. Please refer to the Nucleus FILE target specific notes for your specific driver for more information.

```
// Get the current parameters from the device.
const FMTPARMS* format_parameters = device->GetFormatParameters();

// Format the device using the same format parameters.
device->SetFormatParameters( format_parameters );
device->Format();
```



### Aborting all Pending Operations on a Device

Aborting all operations on the device is accomplished using the `Abort` method. This will cause all resources associated with that drive to be freed, but no disk writes will be attempted. Please note all file descriptors associated with the drive become invalid.

```
// Abort all operations.
device->Abort();

// Go ahead and start over.
device->Open();

// etc.
// ...
```

### Using Directories 1-2-3 (class `NuDirectory`)

The steps and code required for each step are outlined below.

**Please note:** It is important to keep clear in your mind the difference between an object and a physical directory on the device. The object is your programming language interface into Nucleus FILE directory services. The directory is the physical entity on the disk. In the discussions below, we will refer to the *directory object* when talking about the C++ interface and the *directory* when we are referring to the directory on the disk.

### Creating a Directory Object

When you construct a `NuDirectory` directory object, you can optionally pass in a path.

```
NuDirectory directory( "A:\\My_directory" );

NuDirectory another_directory; // Path is empty ("").
```

### Accessing the Path Attributes of a Directory

Path information can be accessed using the `GetPath` and `SetPath` members.

```
// Get the path information from the directory.
const CHAR* path = directory.GetPath();

// Change the path information for this directory.
directory.SetPath( "A:\\Another_Directory" );
```

### Making a Directory on the Device

Creating a directory object is not the same as making a directory on the physical device. The path information currently contained within a directory object may not physically exist on the device. To create the physical directory, you must explicitly call the `Make` member on a directory object that contains the correct path information.

```
directory.Make();
```





## Removing a Directory from the Device

A directory can be removed using the `Remove` member. The directory must be empty.

```
directory.Remove();
```

## Iterating the Contents of a Directory

It is easy to iterate a directory entry on a device using a `NuDirectory` object and the `GetFirst`, `GetNext`, and `Done` members. The following example prints all of the filenames of the files in a directory. It uses a list to hold the `DSTAT` structures that contain the file and sub-directory name information.

```
// Go through the directory and insert copies of DSTAT structures in a list
DSTAT statobj;

// Start the search.
if( directory.GetFirst( &statobj ) )
{
    list<DSTAT> testList;

    do
    {
        // Insert it onto the back of the list.
        testList.push_back( statobj );
    } while( directory.GetNext( &statobj ) );

    // Close the search.
    directory.Done( &statobj );

    // Print the filename of all files in the directory.
    list<DSTAT>::iterator testListIterator = testList.begin();
    while( testListIterator != testList.end() )
    {
        cout << (*testListIterator++).fname; // the filename.
    }

    // Remove the elements from the list.
    testList.erase( testList.begin(), testList.end() );
}
```

## Using Files 1-2-3 (class NuFile)

The steps and code required for each step are outlined below.

**Please note:** It is important to keep clear in your mind the difference between an object and a physical file on the device. The object is your programming language interface into Nucleus FILE services. The file is the physical entity on the disk. In the discussions below, we will refer to the *file object* when talking about the C++ interface and the *file* when we are referring to the file on the disk.

### Creating and Subsequently Opening Files

Creating a file object is different than creating a file on the physical device. You can create a file object that specifies a filename for a file that does not yet exist on the device. Or, you can create a file object using a filename that already exists on the device. Or, you can specify to create a file on the physical device if it does not already exist.

To create a file object, construct a `NuFile` object. The default constructor creates a blank file object that you can later setup. You can optionally specify whether to automatically open the file on the disk.

```
// Create a file object and open the file.
NuFile* file = new NuFile
(
    "My_filename.text",
    (NuFile::readWriteOpen),
    NuFile::writeMode,
    TRUE    // automatically open the file.
);
```

You can tell Nucleus C++ FILE to automatically create a file if it does not already exist using the `NuFile::createOpen` enumerated value.

```
// Create a file object, open the file, and create the file on the
// device if it does not already exist.
NuFile* file = new NuFile
(
    "My_filename.text",
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);
```

### Opening a File on the Device

If you did not instruct the system to open the file when the object was constructed, you must open it using the `Open` member prior to accessing the file on the device. This is another opportunity to rename a file object. Please note you can also specify the file is created on the device if it does not exist. The open flags and mode parameters are the same as the ones used in the constructor.

```
file->Open
(
    "My_filename.text",
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode
);
```

### Closing a File on the Device

After you are complete performing operations on a file, you must close it to return resources to the system.

```
file->Close();
```



## Renaming a File on the Device

There are two forms of rename members. One takes a new string for a file system object. The other is a static class member that takes two strings, one for a source file and the other for a new name of the source file.

```
// Rename a file without an object. Use the class member.
NuFile::Rename( "My_Old_Filename.text", "My_New_Filename.text" );

// Rename a file using an object.
file->Rename( "Another_Filename.text" );
```

## Deleting a File from the Device

There are two forms of delete members. One takes no parameters and deletes the physical file associated with a file object. The other is a static class member that takes a string representing the file on the device to delete.

```
// Delete a file.
file->Delete();

// Delete another file using the class member.
NuFile::Delete( "Another_Filename.text" );
```

## Accessing File Attributes

To gain access to the file descriptor, use `GetDescriptor`.

```
INT file_descriptor = file->GetDescriptor();
```

To gain access to the filename information, use `GetFileName`.

```
CHAR filename [EMAXPATH];
file->GetFileName( filename, EMAXPATH );
```

To gain access to the path information, use `GetPathName`.

```
CHAR path[EMAXPATH];
file-> GetPathName( path, EMAXPATH );
```

To know whether a thread has a file open, use `IsOpen`.

```
if( file->IsOpen() )
{
    // The file is open.
}
```



### Seeking the File Pointer

Before reading or writing to a file, you must first seek the Nucleus FILE pointer location within the file to the byte location you want to read from or write to. There are three forms of seek members in the `NuFile` class. The first two allow you to seek to the beginning and end of a file. The last allows you to seek to an absolute position.

```
// Seek to the end.
file->SeekToEnd();

// Read or write ...

// Seek to the beginning.
file->SeekToBegin();

// Read or write ...

// Seek to exactly the one-kilobyte mark.
file->Seek( 1024, NuFile::beginSeekPosition );

// Read or write ...
```

### Writing to a File

Once you seek to the correct position within a file, you can write bytes to the physical device using the `Write` method.

```
INT32 bytes_written = file->Write( "test_string", sizeof("test_string") );
```

### Reading from a File

Once you seek to the correct position within a file, you can read bytes from the physical device using the `Read` method.

```
CHAR buffer[1024];
INT32 bytes_read = file->Read( buffer, sizeof( buffer ) );
```

### Truncating File Contents

You can truncate a file to any position you specify using the `Truncate` member.

```
// Truncate a file to a length of 1024.
file->Truncate( 1024 );
```

### Flushing the File Buffers

After performing many file operations, you can force the file system device buffers within Nucleus FILE to flush to make sure the actual physical contents of the device are consistent with the file object.

```
file->Flush();
```

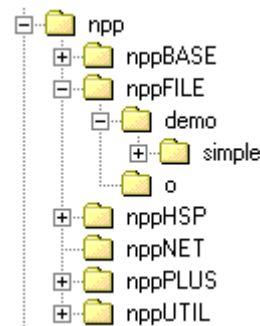


## Nucleus C++ FILE SIMPLE Demonstration Application

The Nucleus C++ FILE demonstration application, `SIMPLE`, demonstrates the basic Nucleus C++ FILE classes `NppFILE`, `NuFileDevice`, `NuDirectory`, and `NuFile`. It creates three identical application tasks and each is parameterized to work out of a separate sub-directory and use unique filenames. You will notice none of the application tasks explicitly block, only the multitasking features of the Nucleus PLUS operating system allow them to execute concurrently.

### Files and File Locations

The Nucleus C++ FILE `SIMPLE` demonstration application source files are located in the `npp\nppFILE\demo\simple` subdirectory. In this directory, you will also find the build files required to build the application using your specific tools. The build files will be named `nppFILE_demo_simple.*` where `*` will be specific to the tools used to create the application. For example, the application that is built using Nucleus EDE has build files named `nppFILE_demo_simple.EDE` and `nppFILE_demo_simple.DSP`.



**The Simple Nucleus C++ FILE Demo Directories**

Source File	Description
<code>NPPAPP.CPP</code>	The main implementation file for the demo, including the initialization points for the application ( <code>NppCreate</code> and <code>NppCreateMultitasking</code> ). If Nucleus C++ PLUS is not present, this file contains all of the implementation.
<code>FILEDEMO.H</code>	The main header file for the demo task classes.
<code>FILEDEMO.INL</code>	The main inline file for the demo task classes.
<code>FILEDEMO.CPP</code>	The main implementation file for the demo task classes.



### Downloading and Executing the Demo

Downloading and executing the SIMPLE application is the same as downloading and executing the Nucleus FILE demonstration application. Please refer to the Nucleus FILE target specific notes for more information.

### What Storage Device is in Use?

The demo is designed to work with the device number that is setup in the file FILEDEMO.CPP using a constant, NPP\_FILE\_DEMO\_DEVICE. It is usually setup to use the RAMDISK in a typical Nucleus FILE system by setting the constant to 0 ("A:"). Please refer to the target specific notes as well as the bottom of file FILEDEMO.CPP to see if the device mapping has changed for your specific hardware configuration.

```
/******  
  
// Change this device mapping if you want to use another device.  
const UINT16 NPP_FILE_DEMO_DEVICE = 0; // ("A:")  
  
/****** end of file *****/
```

### Assertions in Example Code

The demonstration system for Nucleus C++ FILE uses *assertion* macros. These are used to increase readability since every operation is checked for possible errors and the global Nucleus C++ error handling mechanism is invoked upon encountering any. If there is an error, demonstration system threads will spin in `error()` and the string passed will indicate the function call that caused the error.

There are two forms of the assertion macros. The first checks for a good status (STATUS) return value and the other checks for a good Boolean (BOOL) return value:

```
/******  
  
#define ASSERT_STATUS(X,Y)  if((X)!=NU_SUCCESS) error( (const char*)(Y) );  
#define ASSERT_BOOL(X,Y)    if((X)==FALSE) error( (const char*)(Y) );  
  
/******
```

The following code snippet shows a typical call to a routine that checks for status. The first snippet shows the call without using a macro and the second demonstrates the same call using the macro.

```
// Create the directory.  
if( directory.Make() != NU_SUCCESS )  
{  
    error( "directory.Make()" );  
}  
  
// Do the same thing using macros.  
ASSERT_STATUS( directory.Make(), "directory.Make()" )
```



The following code snippet shows a typical call to a routine that checks for Boolean. The first snippet shows the call without using a macro and the second demonstrates the same call using the macro.

```
// Become a file system user.
if( !(nppFILE->BecomeUser()) )
{
    error( "nppFILE->BecomeUser()" );
}

// Do the same thing using macros.
ASSERT_BOOL( nppFILE->BecomeUser(), "NppFILE::BecomeUser()" )
```

## Does Nucleus C++ PLUS Exist?

Nucleus C++ FILE does not require Nucleus C+ PLUS. However, the *SIMPLE* Nucleus C++ FILE demonstration application does take advantage of Nucleus C++ PLUS, if it is present. If it is not present, only a single thread is demonstrated since not taking advantage of the multitasking classes in Nucleus C++ PLUS impedes understanding of the file system due to algorithmic clutter. You will notice a check for the presence of Nucleus C++ PLUS in the demonstration file, `NppAPP.CPP`:

```
// Only use Nucleus C++ PLUS if present.
#if (NPP_PLUS)
    // Nucleus C++ PLUS component.
    NppPLUS* nppPLUS = new NppPLUS;
    nppPLUS->Initialize();
#endif
```

The sections outside of the `#ifdef` blocks are not documented here. However, the behavior of the code is identical to the code inside the derived `Task` object.

## Startup and Initialization

This section discusses the demonstration system startup and initialization.

### System Startup Thread

Since Nucleus FILE must be initialized on a *task* thread as opposed to the *system startup* thread, the *SIMPLE* demo uses `NppCreateMultitasking` as the application initialization point and `NppCreate` is empty.

```
void
NppCreate( void* /* first_available_memory */ )
{
    // Perform all initialization on the multitasking startup thread.
    #if !(NU_APPLICATION_INITIALIZE_MULTITASKING)

        // This demo requires the multitasking startup.
        #error Must have NU_APPLICATION_INITIALIZE_MULTITASKING.

    #endif
}
```



## Multitasking Startup Thread

The `NppCreateMultitasking` initialization routine creates and initializes the various Nucleus C++ components the SIMPLE demo uses: `NppBASE`; `NppPLUS`; `NppFILE`; `NuFileDevice`; and `NppSTATIC`. This initialization sequence relies on the constant setting `NPP_FILE_DEMO_DEVICE`, in the file `FILEDEMO.CPP`.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    /***** begin typical Nucleus C++ initialization ****/

    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    // Only use Nucleus C++ PLUS if present.
    #if (NPP_PLUS)
        // Nucleus C++ PLUS component.
        NppPLUS* nppPLUS = new NppPLUS;
        nppPLUS->Initialize();
    #endif

    // Nucleus C++ FILE component.
    NppFILE* nppFILE = new NppFILE;
    nppFILE->Initialize();

    // Nucleus C++ FILE device.
    NuFileDevice* nppFILE_Device = new NuFileDevice(NPP_FILE_DEMO_DEVICE);
    nppFILE_Device->Initialize();

    // Create a Nucleus C++ standard STREAMS object that is mapped to
    // a physical device in a port specific file.

    #if (NU_STATIC_OBJECT_SUPPORT)
        // ***NOTICE*** if there are any objects declared at file
        // scope, their constructors will be called now.
        NppSTATIC* nppSTATIC = new NppSTATIC;
        nppSTATIC->Initialize();
    #endif

    /***** end typical Nucleus C++ initialization ****/

    /***** Demonstration application *****/
    ... the application initialization code is described below.
}
```

You will notice `NppSTATIC` is the last Nucleus C++ support component initialized prior to any application code. This is done since `NppSTATIC` is responsible for iterating the list of static objects within the system and calls the constructors of these objects one-by-one. Since the `NppFILE` component has already been initialized, full file system facilities are available to all static objects within the system, including services that are accessed within constructors.





## Streaming Statistical Output

The Nucleus C++ FILE demonstration tasks update their statistics in a derived NppStreamable statistics class, FileTestStatistics. For a more detailed discussion of the standard Nucleus C++ demonstration system output facilities, please refer to the Nucleus C++ BASE User's Guide.

### class FileTestStatistics

The instance of each statistics object is encapsulated within the Nucleus C++ FILE demonstration system task class. The constructors of these tasks register their statistics objects with the supplied streaming output task. The FileTestStatistics class uses an NppProtect object to manage reentrancy and synchronization between the output task and the file system task.

```
class FileTestStatistics : public NppStreamable
{
public:

    FileTestStatistics( const CHAR* init_name );

    virtual
    ~FileTestStatistics();

    // Streams this object to the supplied ostream.
    virtual
    void
    StreamToOstream( ostream_Npp& os );

    void    IncrementFilesCreated();
    void    IncrementFilesRenamed();
    void    IncrementFilesDeleted();
    void    IncrementDirectoriesCreated();
    void    IncrementDirectoriesDeleted();

protected:

    NppProtect    protect;
    INT           files_created;
    INT           files_renamed;
    INT           files_deleted;
    INT           directories_created;
    INT           directories_deleted;
    const CHAR*   name;
};
```



### Startup and Initialization

The following code shows how the output task is setup in `NppCreateMultitasking`.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    /***** Nucleus C++ components *****/
    ... Nucleus C++ component initialization is described above.

    /***** Output stream setup *****/

    // Use Nucleus C++ Singleton cout for all demo output.
    ostream_Npp& cout_Npp = *NppStandardStreamsComponent()->cout();

    // Put out the release banner.
    cout_Npp    << endl
                << endl
                << BannerLine << endl
                << "Hello Nucleus C++ FILE world!" << endl
                << endl
                << ReleaseInformation << endl
                << BannerLine << endl
                << endl
                << endl;

    // Use cout and refresh every 2 seconds.
    DemoStreamingTask* streamingTask = new DemoStreamingTask( cout_Npp, 2 );

    // Start the output task.
    streamingTask->Start();

    // Force 4 endlines onto the data stream.
    streamingTask->Register(new DemoStreamableEndline(new NppStreamable,4));

    /***** Rest of demonstration application *****/
    ... the application initialization code is described below.
}
```

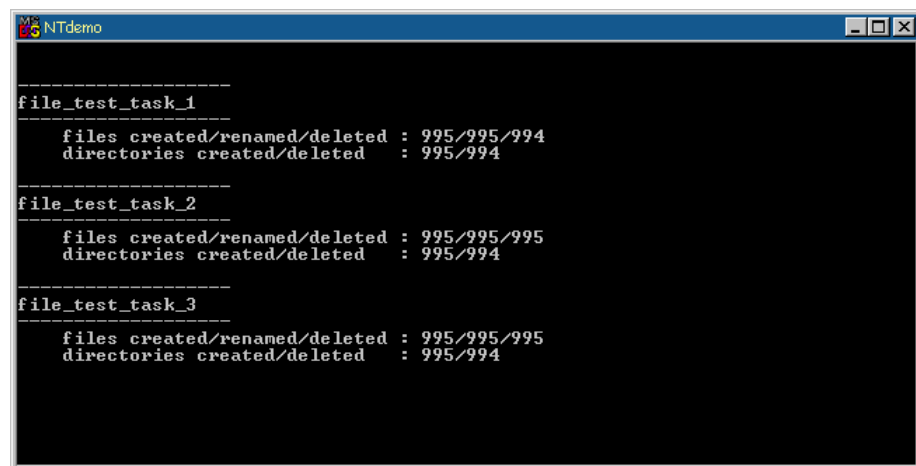


## StreamToOstream Method

The output method simply outputs the statistics. Since the demo creates three tasks, there will be three calls to this method, one call for each statistic object that was registered.

```
void
FileTestStatistics::StreamToOstream( ostream_Npp& os )
{
    os << QuarterBannerLine
      << endl
      << name
      << endl
      << QuarterBannerLine
      << endl
      << Indent
      << "files created/renamed/deleted : "
      << protect.GetProtectedINT( files_created )
      << "/"
      << protect.GetProtectedINT( files_renamed )
      << "/"
      << protect.GetProtectedINT( files_deleted )
      << endl
      << Indent
      << "directories created/deleted : "
      << protect.GetProtectedINT( directories_created )
      << "/"
      << protect.GetProtectedINT( directories_deleted )
      << endl
      << endl;
}
```

The output for the demo is shown below:



```
NTdemo

-----
file_test_task_1
-----
files created/renamed/deleted : 995/995/994
directories created/deleted : 995/994

-----
file_test_task_2
-----
files created/renamed/deleted : 995/995/995
directories created/deleted : 995/994

-----
file_test_task_3
-----
files created/renamed/deleted : 995/995/995
directories created/deleted : 995/994
```

**Nucleus C++ FILE SIMPLE demo output**



## The Demonstration System Task

The following sections document the details of the Nucleus C++ FILE demonstration system task. This task is only used if Nucleus C++ PLUS exists. If it does not exist, the algorithm encapsulated by this class is used in the single multitasking startup thread, `NppCreateMultitasking`, located in the file `NppAPP.CPP`.

### class FileTestTask

The class is shown below. This is a very simple derived `Task` class that takes a task name for reporting purposes, a filename to use, a directory name to use, a pointer to the buffer used for writing, reading, and comparing, a device number, and a pointer to a streaming task for output. Internally, this class has data members that keep track of construction parameters and a statistics object for tracking and outputting results.

```
class FileTestTask : public Task
{
public:
    FileTestTask
    (
        const CHAR*      name,
        const CHAR*      init_filename,
        const CHAR*      init_directoryname,
        const CHAR*      init_test_buffer,
        UINT16           init_device_number,
        DemoStreamingTask* streamingTask
    );

    virtual
    ~FileTestTask();

protected:
    NppFILE*            nppFILE;
    NuFileDevice*       device;
    FileTestStatistics  statistics;
    const CHAR*         filename;
    const CHAR*         directoryname;
    const CHAR*         test_buffer;
    UINT16              device_number;

    virtual
    void
    Entry();
};
```



## Application Startup and Initialization

The instances of class `FileTestTask` are created in `NppCreateMultitasking`. Each task is of course identical, only the construction parameters are different.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    /***** Nucleus C++ components *****/
    ... Nucleus C++ component initialization is described above.

    /***** Demonstration application *****/

    // Create the test tasks and start them.
    FileTestTask* file_test_task_3 = new FileTestTask
    (
        "file_test_task_3",                // task.
        "file_test_task_3.testers",        // file.
        "file_test_task_3_directory",      // directory.
        "this is the test buffer for test task 3", // buffer.
        NPP_FILE_DEMO_DEVICE,              // device number.
        streamingTask                       // output task.
    );
    file_test_task_3->Start();

    FileTestTask* file_test_task_2 = new FileTestTask
    (
        "file_test_task_2",                // task.
        "file_test_task_2.testers",        // file.
        "file_test_task_2_directory",      // directory.
        "this is the test buffer for test task 2", // buffer.
        NPP_FILE_DEMO_DEVICE,              // device number.
        streamingTask                       // output task.
    );
    file_test_task_2->Start();

    FileTestTask* file_test_task_1 = new FileTestTask
    (
        "file_test_task_1",                // task.
        "file_test_task_1.testers",        // file.
        "file_test_task_1_directory",      // directory.
        "this is the test buffer for test task 1", // buffer.
        NPP_FILE_DEMO_DEVICE,              // device number.
        streamingTask                       // output task.
    );
    file_test_task_1->Start();
}
```

### Entry Member

The details of the task's behavior are found within the `Entry` member. The pseudo code is shown below, followed by a discussion of the details.

```
void
FileTestTask::Entry()
{
    // Create a scratch compare buffer and null terminate it.

    // Loop forever.
    while( 1 )
    {
        // Become a user.

        // Open the device for use.
        // Make this the default device.

        // Create the directory.
        // Set the working directory for this thread.

        // Create the file.

        // Seek to the beginning and write in block mode.

        // Seek back to the beginning and read it in.

        // Rename the file.

        // Delete the file.

        // Go back to the root directory for the working directory.

        // Remove the directory.

        // Close the device.

        // Release the user.
    }
}
```

#### Create a scratch compare buffer.

The scratch data is allocated on the stack, at the top of the `Entry` member. This is just a buffer that is the same size as the test buffer supplied in the constructor (the data written).

```
// Create a scratch compare buffer and null terminate it.
INT  buffer_length = strlen( test_buffer );
CHAR* buffer = new CHAR[buffer_length + 1];
buffer[buffer_length] = 0;
```



### Become a user.

Registering as a file system user is accomplished by calling the `BecomeUser` method of the Nucleus C++ FILE component class.

```
// Become a user.
ASSERT_BOOL( nppFILE->BecomeUser(), "NppFILE::BecomeUser()" )
```

### Open the device for use.

A pointer to the device is obtained using a class member of class `NuFileDevice`. Once we have a pointer, we can open the device and set it as the default device for this thread.

```
// Get a pointer to the device.
device = NuFileDevice::GetDevice( device_number );
ASSERT_BOOL( NULL != device, "NULL != device" )

// Open the device for use.
ASSERT_STATUS( device->Open(), "device->Open()" )

// Make this the default device.
ASSERT_STATUS( device->SetDefaultDevice(), "device->SetDefaultDevice()" )
```

### Create the directory.

Creating a directory is easy, just execute the `Make` member of the directory class. The path is passed to the `SetCurrentDirectory` member to set it up as the working directory for this thread.

```
// Create the directory.
NuDirectory directory( directoryname );
ASSERT_STATUS( directory.Make(), "directory.Make()" )

// Update statistics.
statistics.IncrementDirectoriesCreated();

// Set the working directory for this thread.
ASSERT_STATUS
(
    device->SetCurrentDirectory( directory.GetPath() ),
    "device->SetCurrentDirectory( directory.GetPath() )"
)
```

### Create the file.

The file is created and opened in one step in the constructor of the NuFile object.

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

// Make sure it was opened.
ASSERT_BOOL( file->IsOpen(), "file->IsOpen()" )

// Update statistics.
statistics.IncrementFilesCreated();
```

### Seek to the beginning and write.

SeekToBegin is used to seek to the beginning of the file prior to the write operation, which simply uses the Write member of class NuFile.

```
// Seek to the beginning and write in block mode.
ASSERT_BOOL( 0 == file->SeekToBegin(), "0 == file->SeekToBegin()" )

ASSERT_BOOL
(
    buffer_length == file->Write( test_buffer, buffer_length ),
    "buffer_length == file->Write( test_buffer, buffer_length )"
)
```

### Seek back to the beginning and read.

Seek is used to seek to the beginning of the file prior to the read operation, which simply uses the Read member of class NuFile.

```
// Seek back to the beginning and read it in. This time, demonstrate
// absolute seek when we seek to the start of the file.
ASSERT_BOOL
(
    0 == file->Seek( 0, NuFile::beginSeekPosition ),
    "0 == file->Seek( 0, NuFile::beginSeekPosition )"
)

ASSERT_BOOL
(
    buffer_length == file->Read( buffer, buffer_length ),
    "buffer_length == file->Read( buffer, buffer_length )"
)

// Check it.
ASSERT_BOOL
(
    !strcmp((const char*)buffer, (const char*)test_buffer, buffer_length),
    "!strcmp((const char*)buffer, (const char*)test_buffer, buffer_length)"
)
```





### Rename the file.

Renaming the file is easy using the `Rename` method.

```
// Rename the file.
ASSERT_STATUS( file->Rename("test.rename"), "file->Rename(\"test.rename\")" )

// Update statistics.
statistics.IncrementFilesRenamed();
```

### Delete the file.

Deleting the physical file on the disk is not the same as deleting the object. They are two separate steps. Deleting the physical file is accomplished using the `Delete` member, deleting the object is done using the `delete` operator.

```
// Delete the file.
ASSERT_STATUS( file->Delete(), "file->Delete()" )
delete file;

// Update statistics.
statistics.IncrementFilesDeleted();
```

### Go back to the root directory.

Going back to the root is done by passing in a root directory string.

```
// Go back to the root directory for the working directory for this thread..
ASSERT_STATUS
(
    device->SetCurrentDirectory( "\\\" ),
    "device->SetCurrentDirectory( \"\"\\\" )"
)
```

### Remove the directory.

Removing the directory from the disk is different than deleting the directory object. Removing the physical directory is done using the `Remove` member of the directory class.

```
// Remove the directory.
ASSERT_STATUS( directory.Remove(), "directory.Remove()" )

// Update statistics.
statistics.IncrementDirectoriesDeleted();
```

### Close the device.

The device is closed using the `Close` member.

```
// Close the device.  
ASSERT_STATUS( device->Close(), "device->Close()" )
```

### Release the user.

Un-registering as a file system user is accomplished by calling the `ReleaseUser` method of the Nucleus C++ FILE component class.

```
// Release file component resources.  
nppFILE->ReleaseUser();
```



2

# Nucleus C++ FILE Class Reference



### class NuFile

NuFile objects encapsulate a file on a storage device.

#### Public Member Functions

Member	Overview
NuFile	Constructor
~NuFile	Destructor
Open	Opens the file for use.
IsOpen	Returns TRUE if the file is open for use.
Close	Closes a file for use.
SeekToEnd	Seeks the file pointer to the end of the file.
SeekToBegin	Seeks the file pointer to the beginning of the file.
Seek	Seeks the file pointer to the specified position within the file.
Read	Reads bytes from the file from the current file position.
Write	Writes bytes to the file at the current file position.
Truncate	Truncates the file at the specified file position.
Flush	Flushes the internal Nucleus FILE buffers so the disk contents are consistent with the program.
Rename	Renames the file.
GetFileName	Returns the filename portion of the absolute filename.
GetPathName	Returns the path portion of the absolute filename.
Delete	Deletes the file from the disk.
GetDescriptor	Returns the Nucleus FILE descriptor for the file.

#### Public Class Member Functions

Member	Overview
Rename	Renames the file specified to the new name specified. This does not need an object to operate on.
Delete	Deletes the file specified. This does not need an object to operate on.



**NuFile::NuFile**`NuFile();`

Default Constructor. **NOTE:** See second, non-default, constructor below.

**Overview**

Condition	Description
Pre-condition	None
Action	Creates a file object.
Post-condition	The file object exists.

**Parameters**

None

**Return Value**

None

**Example**

```
// Create the file.  
NuFile* file = new NuFile;
```



## NuFile::NuFile

```
NuFile
(
    const CHAR*    initFileName,
    INT            initOpenFlag = (readWriteOpen | createOpen |
                                exclusiveOpen ),
    INT            initOpenMode = writeMode,
    BOOL           bOpen = TRUE
);
```

Constructor

### Overview

Condition	Description
Pre-condition	None
Action	Creates a file object.
Post-condition	The file object exists.

### Parameters

Parameter	Overview
initFileName	The filename.
initOpenFlag	<p>The “open flag” and will be one or more of the following enumerated values (enum NuFile::OpenFlag):</p> <pre>enum OpenFlag {     // Open for read only. readOnlyOpen     = PO_RDONLY,     // Open for write only.     writeOnlyOpen      = PO_WRONLY,     // Read/write access allowed.     readWriteOpen      = PO_RDWR,     // Seek to eof on each write.     appendOpen         = PO_APPEND,     // Create the file if it does not exist.     createOpen         = PO_CREAT,     // Truncate the file if it exists.     truncateOpen       = PO_TRUNC,     // Fail if creating and already exists.     exclusiveOpen      = PO_EXCL,     // Ignored.     textOpen           = PO_TEXT,     // Ignored. All file access is binary.     binaryOpen         = PO_BINARY,     // Wants this open to fail if already     // open. Other opens will fail while     // this open is active.     noShareOpen        = PO_NOSHAREANY,     // Wants this opens to fail if already     // open for write. Other open for     // write calls will fail while this     // open is active.     noShareWriteOpen   = PO_NOSHAREWRITE };</pre>



initOpenMode	<p>The “open mode” and will be one or more of the following enumerated values (enum NuFile::OpenMode):</p> <pre>enum OpenMode {     // Write permitted.     writeMode    = PS_IWRITE,     // Read permitted. (Always true anyway).     readMode     = PS_IREAD };</pre>
bOpen	TRUE if the file is to be automatically opened.

**Return Value**

None

**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);
```

### NuFile::~~NuFile

```
virtual  
~NuFile();
```

Destructor

#### Overview

Condition	Description
Pre-condition	None
Action	Destructs the object and returns resources to the system.
Post-condition	The object does not exist.

#### Parameters

None

#### Return Value

None

#### Example

Destructors are called automatically when the object goes out of scope or is deleted.





**NuFile::Open**

```
virtual  
STATUS  
Open();
```

Opens the file for use.

**Overview**

Condition	Description
Pre-condition	The file is not open.
Action	Opens the file within Nucleus FILE.
Post-condition	The file is open.

**Parameters**

None



**Return Value**

Return Value	Overview
STATUS	<p>Returns a non-negative integer to be used as a file descriptor. Returns a negative integer indicates an error as follows:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_NOSPC – No space to create directory</p> <p>NUF_LONGPATH – Path or directory name too long.</p> <p>NUF_INVNAME – Path or filename includes invalid character.</p> <p>NUF_INVPARM – Invalid Flag/Mode is specified.</p> <p>NUF_INVPARCMB – Invalid Flag/Mode combination</p> <p>NUF_PEMFILE – No file descriptors available. (Too many files open)</p> <p>NUF_ACCES – You can't open the file that has Directory or VOLLABEL attributes.</p> <p>NUF_NOSPC – No space to create directory in this disk.</p> <p>NUF_SHARE – The access conflict from multiple tasks to a specific file.</p> <p>NUF_NOFILE – The specified file not found.</p> <p>NUF_EXIST – The directory already exists.</p> <p>NUF_NO_BLOCK – No block buffer available.</p> <p>NUF_NO_FINODE – No FINODE buffer available.</p> <p>NUF_NO_DROBJ – No DROBJ buffer available.</p> <p>NUF_IO_ERROR – IO error occurred.</p> <p>NUF_INTERNAL – Nucleus FILE internal error. fs_user-&gt;p_errno is set to one of these values</p> <p>PENOENT – File not found or path to file not found.</p> <p>PEMFILE – No file descriptors available (too many files open).</p> <p>PEEXIST – Exclusive access requested but file already exists.</p> <p>PEACCES – Attempt to open a read only file or a special (directory) file.</p> <p>PENOSPC – Create failed.</p> <p>PESHARE – Already open in exclusive mode or we want exclusive and it's already open.</p>



**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    FALSE // DO NOT open the file.
);

if( file->Open() == NU_SUCCESS )
{
    // The file was open OK.
}
```

### NuFile::Open

```
virtual
INT
Open
(
    const CHAR*    initFileName,
    INT            initOpenFlag = (readWriteOpen | createOpen |
                                   exclusiveOpen ),
    INT            initOpenMode = writeMode
);
```

Opens the file for use.

### Overview

Condition	Description
Pre-condition	The file is not open.
Action	Opens the file within Nucleus FILE.
Post-condition	The file is open.



**Parameters**

Parameter	Overview
initFileName	The filename.
initOpenFlag	<p>The “open flag” and will be one or more of the following enumerated values (enum NuFile::OpenFlag):</p> <pre> enum OpenFlag {     // Open for read only. readOnlyOpen    = PO_RDONLY,     // Open for write only.     writeOnlyOpen    = PO_WRONLY,     // Read/write access allowed.     readWriteOpen    = PO_RDWR, // Seek to eof on each                                 write.     appendOpen= PO_APPEND, // Create the file if it                                 does not exist.     CreateOpen = PO_CREAT, // Truncate the file if it                                 exists.     TruncateOpen = PO_TRUNC, // Fail if creating and                                 already exists.     exclusiveOpen    = PO_EXCL, // Ignored.     textOpen         = PO_TEXT, // Ignored. All file                                 access is binary.      binaryOpen       = PO_BINARY,     // Wants this open to fail if already     // open. Other opens will fail while     // this open is active.     NoShareOpen = PO_NOSHAREANY,     // Wants this opens to fail if already     // open for write. Other open for     // write calls will fail while this     // open is active.     noShareWriteOpen    = PO_NOSHAREWRITE }; </pre>
initOpenMode	<p>The “open mode” and will be one or more of the following enumerated values (enum NuFile::OpenMode):</p> <pre> enum OpenMode {     // Write permitted.     writeMode        = PS_IWRITE,     // Read permitted. (Always true anyway).     readMode         = PS_IREAD }; </pre>

**Return Value**

Return Value	Overview
INT	<p>Returns a non-negative integer to be used as a file descriptor. Returns a negative integer indicates an error as follows:</p> <p>NUF_BAD_USER – Not a file user.  NUF_BADDRIVE – Invalid drive specified.  NUF_NOSPC – No space to create directory  NUF_LONGPATH – Path or directory name too long.  NUF_INVNAME – Path or filename includes invalid character.  NUF_INVPARM – Invalid Flag/Mode is specified.  NUF_INVPARCMB – Invalid Flag/Mode combination  NUF_PEMFILE – No file descriptors available.  (Too many files open)  NUF_ACCES – You can't open the file, which has Directory or VOLLABEL attributes.  NUF_NOSPC – No space to create directory in this disk.  NUF_SHARE – The access conflict from multiple tasks to a specific file.  NUF_NOFILE – The specified file not found.  NUF_EXIST – The directory already exists.  NUF_NO_BLOCK – No block buffer available.  NUF_NO_FINODE – No FINODE buffer available.  NUF_NO_DROBJ – No DROBJ buffer available.  NUF_IO_ERROR – IO error occurred.  NUF_INTERNAL – Nucleus FILE internal error.  fs_user-&gt;p_errno is set to one of these values  PENOENT – File not found or path to file not found.  PEMFILE – No file descriptors available (too many files open).  PEEXIST – Exclusive access requested but file already exists.  PEACCES – Attempt to open a read only file or a special (directory) file.  PENOSPC – Create failed.  PESHARE – Already open in exclusive mode or we want exclusive and it's already open.</p>



**Example**

```
// Create the file.
NuFile* file = new NuFile;

if
(
    file->Open()
    (
        filename,
        (NuFile::readWriteOpen|NuFile::createOpen),
        NuFile::writeMode
    )
    ==
    NU_SUCCESS
)
{
    // The file was open OK.
}
```

### NuFile::IsOpen

virtual

BOOL

IsOpen() const;

Returns TRUE if the file is open for use.

### Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

### Parameters

None

### Return Value

Return Value	Overview
BOOL	Returns TRUE if the file is open for use.

### Example

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

// Make sure it was opened.
if( file->IsOpen() )
{
    // The file was opened OK.
}
```





**NuFile::Close**

```
virtual
STATUS
Close();
```

Closes a file for use.

**Overview**

Condition	Description
Pre-condition	The file is open.
Action	Closes the file within Nucleus FILE.
Post-condition	The file is closed.

**Parameters**

None

**Return Value**

Return Value	Overview
STATUS	<p>Returns the status of the method:</p> <p>NU_SUCCESS – Close the file was successfully.</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADFILE – Invalid file descriptor.</p> <p>NUF_IO_ERROR – IO error occurred.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

// Operate on the file.....

// Close it.
if( file->Close == NU_SUCCESS )
{
    // The file was closed OK.
}
```

### NuFile::SeekToEnd

```
virtual  
INT32  
SeekToEnd();
```

Seeks the file pointer to the end of the file.

#### Overview

Condition	Description
Pre-condition	The file is open.
Action	Seeks the file pointer to the end position.
Post-condition	The file pointer is at the end position.

#### Parameters

None

#### Return Value

Return Value	Overview
INT32	Returns a non-negative integer to be used as a number of bytes seek. Otherwise:  NUF_BAD_USER – Not a file user. NUF_BADPARAM – Invalid parameter specified. NUF_BADFILE – File descriptor invalid. NUF_IO_ERROR – Driver IO error. NUF_INTERNAL – Nucleus FILE internal error.

#### Example

```
// Create the file.  
NuFile* file = new NuFile  
(  
    filename,  
    (NuFile::readWriteOpen|NuFile::createOpen),  
    NuFile::writeMode,  
    TRUE    // open the file.  
);  
  
INT32 end_seek_position = file->SeekToEnd();
```



**NuFile::SeekToBegin**

```
virtual
INT32
SeekToBegin();
```

Seeks the file pointer to the beginning of the file.

**Overview**

Condition	Description
Pre-condition	The file is open.
Action	Seeks the file pointer to the begin position.
Post-condition	The file pointer is at the begin position.

**Parameters**

None

**Return Value**

Return Value	Overview
INT32	<p>Returns a non-negative integer to be used as a number of bytes seek. Otherwise:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADPARAM – Invalid parameter specified.</p> <p>NUF_BADFILE – File descriptor invalid.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

if( 0 == file->SeekToBegin() )
{
    // Theseek to begin (0) worked.
}
```

## NuFile::Seek

```
virtual
INT32
Seek( INT32 offset, NuFile::SeekPosition seekPosition );
```

Seeks the file pointer to the specified position within the file.

### Overview

Condition	Description
Pre-condition	The file is open.
Action	Seeks the file pointer to the begin position.
Post-condition	The file pointer is at the begin position.

### Parameters

Parameter	Overview
offset	The seek position, in bytes, relative to the next parameter, the seek position.
seekPosition	<p>The “seek position” and will be one or more of the following enumerated values (enum NuFile::OpenFlag):</p> <pre>enum SeekPosition {     // offset from beginning of file.     beginSeekPosition    = PSEEK_SET,     // offset from current file pointer.     currentSeekPosition  = PSEEK_CUR,     // offset from end of file.     endSeekPosition      = PSEEK_END };</pre>

### Return Value

Return Value	Overview
INT32	<p>Returns a non-negative integer to be used as a number of bytes seek. Otherwise:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADPARM – Invalid parameter specified.</p> <p>NUF_BADFILE – File descriptor invalid.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>



**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

INT32 seek_position = 1024;
seek_position = file->Seek( NuFile::beginSeekPosition );
```

### NuFile::Read

```
virtual  
INT32  
Read( CHAR* buffer, INT32 count );
```

Reads bytes from the file from the current file position.

#### Overview

Condition	Description
Pre-condition	The file is open.
Action	The file is read from.
Post-condition	The buffer contains the number of characters read from the file.

#### Parameters

Parameter	Overview
buffer	A pointer to the buffer that will receive the characters read.
count	The number of bytes to read into the buffer.

#### Return Value

Return Value	Overview
INT32	Returns the number of bytes read or negative value on error. Otherwise, it returns a negative error code:  NUF_BAD_USER – Not a file user. NUF_BADPARM – Invalid parameter specified. NUF_BADFILE – Invalid file descriptor. NUF_ACCES – Open flag is PO_WRONLY. NUF_IO_ERROR – IO error occurred. NUF_INTERNAL – Nucleus FILE internal error.



**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

if( 0 == file->SeekToBegin() )
{
    // The seek to begin (0) worked.
}

CHAR buffer[1024];
INT32 number_bytes_read = file->Read( buffer, 1024 );
```

### NuFile::Write

```
virtual  
INT32  
Write( const CHAR* buffer, INT32 count );
```

Writes bytes to the file at the current file position.

#### Overview

Condition	Description
Pre-condition	The file is open.
Action	The file is written to.
Post-condition	The file has been written to.

#### Parameters

Parameter	Overview
buffer	A pointer to the buffer that holds the characters to write.
count	The number of bytes to write from the buffer.

#### Return Value

Return Value	Overview
INT32	Returns the number of bytes written or negative value on error. If the return value is negative, the meaning is follows:  NUF_BAD_USER – Not a file user. NUF_BADPARM – Invalid parameter specified. NUF_BADFILE – File descriptor invalid. NUF_ACCES – Not a PO_WRONLY or PO_RDWR open flag or file attributes is ARDONLY. NUF_NOSPC – Write failed. Presumably because of no space. NUF_IO_ERROR – IO error occurred. NUF_INTERNAL – Nucleus FILE internal error.

#### Example

```
// Create the file.  
NuFile* file = new NuFile  
(  
    filename,  
    (NuFile::readWriteOpen|NuFile::createOpen),  
    NuFile::writeMode,  
    TRUE    // open the file.  
);  
if( 0 == file->SeekToBegin() )  
{  
    // The seek to begin (0) worked.  
}  
CHAR buffer = "hello";  
INT32 number_bytes_written = file->Write( buffer, strlen(buffer) );
```





**NuFile::Truncate**

```
virtual
STATUS
Truncate( INT32 offset );
```

Truncates the file at the specified file position.

**Overview**

Condition	Description
Pre-condition	The file is open.
Action	Truncates the file.
Post-condition	The file is truncated or an error is returned.

**Parameters**

Parameter	Overview
offset	The byte position to truncate the file at.

**Return Value**

Return Value	Overview
STATUS	<p>Returns NU_SUCCESS if the truncate the data was successful. Otherwise, it returns the following error code:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADPARM – Invalid parameter specified.</p> <p>NUF_BADFILE – File descriptor invalid.</p> <p>NUF_ACCES – You can't change the file, which has PO_RDONLY, or file attributes are ARDONLY.</p> <p>NUF_SHARE – The access conflict from multiple tasks to a specific file.</p> <p>NUF_NO_BLOCK – No block buffer available.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

// Truncate the file to 1024 bytes in length.
if( NU_SUCCESS == file->Truncate( 1024 ) )
{
    // The file was successfully truncated.
}
```



### NuFile::Flush

```
virtual  
STATUS  
Flush();
```

Flushes the internal Nucleus FILE buffers so the disk contents are consistent with the program.

#### Overview

Condition	Description
Pre-condition	The file is open.
Action	Flushes the internal Nucleus FILE buffers.
Post-condition	The disk contents are consistent with the program.

#### Parameters

None

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS upon success. Otherwise one of the following error codes will be returned:  NUF_BAD_USER – Not a file user. NUF_BADFILE – Invalid file descriptor. NUF_IO_ERROR – IO error occurred. NUF_INTERNAL – Nucleus FILE internal error.

#### Example

```
// Create the file.  
NuFile* file = new NuFile  
(  
    filename,  
    (NuFile::readWriteOpen|NuFile::createOpen),  
    NuFile::writeMode,  
    TRUE    // open the file.  
);  
  
if( 0 == file->SeekToBegin() )  
{  
    // The seek to begin (0) worked.  
}  
  
CHAR buffer = "hello";  
INT32 number_bytes_written = file->Write( buffer, strlen(buffer) );  
  
// Flush the buffers to make the disk consistent with this program.  
if( NU_SUCCESS == file->Flush() )  
{  
    // The file was successfully flushed.  
}
```



**NuFile::Rename**

```
virtual
INT
Rename( const CHAR* newName );
```

Renames the file.

**Overview**

Condition	Description
Pre-condition	The file exists.
Action	The file is renamed.
Post-condition	The file has been renamed.

**Parameters**

Parameter	Overview
NewName	The new name of the file.

**Return Value**

Return Value	Overview
INT	<p>Returns NU_SUCCESS upon success. Otherwise, one of the following error codes is returned:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_BADPARM – Invalid parameter specified.</p> <p>NUF_ROOT_FULL – Root directory full.</p> <p>NUF_INVNAME – Path or filename includes invalid character.</p> <p>NUF_ACCES – You can't change the file which has VOLLABEL, HIDDEN. Specify newname path.</p> <p>NUF_NOSPC – No space to create directory</p> <p>NUF_NOFILE – The specified file not found.</p> <p>NUF_NO_BLOCK – No block buffer available.</p> <p>NUF_NO_FINODE – No FINODE buffer available.</p> <p>NUF_NO_DROBJ – No DROBJ buffer available.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

### Example

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

// Rename it.
if( NU_SUCCESS == file->Rename( "newfilename.text" ) )
{
    // The file rename was successful.
}
```



**NuFile::Rename**

```
static
INT
Rename( const CHAR* oldname, const CHAR* newName );
```

Renames the file.

**Overview**

Condition	Description
Pre-condition	The file exists.
Action	The file is renamed.
Post-condition	The file has been renamed.

**Parameters**

Parameter	Overview
OldName	The old name of the file.
NewName	The new name of the file.

**Return Value**

Return Value	Overview
INT	<p>Returns NU_SUCCESS upon success. Otherwise, one of the following error codes is returned:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_BADPARM – Invalid parameter specified.</p> <p>NUF_ROOT_FULL – Root directory full.</p> <p>NUF_INVNAME – Path or filename includes invalid character.</p> <p>NUF_ACCES – You can't change the file which has VOLLABEL, HIDDEN. Specify newname path.</p> <p>NUF_NOSPC – No space to create directory.</p> <p>NUF_NOFILE – The specified file not found.</p> <p>NUF_NO_BLOCK – No block buffer available.</p> <p>NUF_NO_FINODE – No FINODE buffer available.</p> <p>NUF_NO_DROBJ – No DROBJ buffer available.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

**Example**

```
// Rename the file.
if( NU_SUCCESS == NuFile::Rename("oldfilename.text", "newfilename.text") )
{
    // The file rename was successful.
}
```



### NuFile::GetFileName

```
virtual  
STATUS  
GetFileName( CHAR* name, INT maxNameSize ) const;
```

Returns the filename portion of the absolute filename.

#### Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

#### Parameters

Parameter	Overview
name	The destination buffer that will receive the filename.
maxNameSize	The maximum number of characters to copy into the destination.

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS if the operation was successful, otherwise one of the following error codes is returned:  NUF_LONGPATH – Path or filename too long. NUF_INVNAME – Path or filename includes invalid character.

#### Example

```
// Create the file.  
NuFile* file = new NuFile  
(  
    filename,  
    (NuFile::readWriteOpen|NuFile::createOpen),  
    NuFile::writeMode,  
    TRUE    // open the file.  
);  
  
CHAR filename_destination [32];  
if( NU_SUCCESS == file->GetFileName( filename_destination, 32 ) )  
{  
    // The operation was successful.  
}
```



**NuFile::GetPathName**

virtual

STATUS

GetPathName( CHAR\* name, INT maxNameSize ) const;

Returns the path portion of the absolute filename.

**Overview**

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

**Parameters**

Parameter	Overview
Name	The destination buffer that will receive the path name.
maxNameSize	The maximum number of characters to copy into the destination.

**Return Value**

Return Value	Overview
STATUS	Returns NU_SUCCESS if the operation was successful, otherwise one of the following error codes is returned: NUF_LONGPATH – Path or filename too long. NUF_INVNAME – Path or filename includes invalid character.

**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    TRUE    // open the file.
);

CHAR pathname_destination [32];
if( NU_SUCCESS == file->GetPathName( pathname_destination, 32 ) )
{
    // The operation was successful.
}
```

### NuFile::Delete

```
virtual  
STATUS  
Delete();
```

Deletes the file from the disk.

#### Overview

Condition	Description
Pre-condition	The file exists.
Action	The file is deleted from the disk.
Post-condition	The file has been deleted from the disk.

#### Parameters

None

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS if the operation was successful, otherwise one of the following error codes is returned:  NUF_BAD_USER – Not a file user. NUF_BADDRIVE – Invalid drive specified. NUF_INVNAME – Path or filename includes invalid character. NUF_ACCES – This file has at least one of the following attributes: RDONLY, HIDDEN, SYSTEM, VOLUME, DIRENT. NUF_SHARE – The access conflict from multiple tasks to a specific file. NUF_NOFILE – The specified file not found. NUF_NO_BLOCK – No block buffer available. NUF_NO_FINODE – No FINODE buffer available. NUF_NO_DROBJ – No DROBJ buffer available. NUF_IO_ERROR – Driver IO function routine NUF_INTERNAL – Nucleus FILE internal error.





**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    FALSE // do not open the file.
);

// Delete the file.
if( NU_SUCCESS == file->Delete() )
{
    // The file was successfully deleted.
}
```

### NuFile::Delete

```
static  
STATUS  
Delete( const CHAR* fileName );
```

Deletes the file from the disk.

### Overview

Condition	Description
Pre-condition	The file exists.
Action	The file is deleted from the disk.
Post-condition	The file has been deleted from the disk.

### Parameters

Parameter	Overview
filename	The filename of the file to delete.

### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS if the operation was successful, otherwise one of the following error codes is returned: NUF_BAD_USER – Not a file user. NUF_BADDRIVE – Invalid drive specified. NUF_INVNAME – Path or filename includes invalid character. NUF_ACCES – This file has at least one of the following attributes: RDONLY, HIDDEN, SYSTEM, VOLUME, DIRENT. NUF_SHARE – The access conflict from multiple tasks to a specific file. NUF_NOFILE – The specified file not found. NUF_NO_BLOCK – No block buffer available. NUF_NO_FINODE – No FINODE buffer available. NUF_NO_DROBJ – No DROBJ buffer available. NUF_IO_ERROR – Driver IO function routine NUF_INTERNAL – Nucleus FILE internal error.

### Example

```
// Delete the file.  
if( NU_SUCCESS == NuFile::Delete( "my_filename.text" ) )  
{  
    // The file was successfully deleted.  
}
```



**NuFile::GetDescriptor**

```
virtual
INT
GetDescriptor();
```

Returns the Nucleus FILE descriptor for the file.

**Overview**

Condition	Description
Pre-condition	The file has been opened.
Action	None
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
INT	A valid, non-negative file descriptor or -1 if the file is not open.

**Example**

```
// Create the file.
NuFile* file = new NuFile
(
    filename,
    (NuFile::readWriteOpen|NuFile::createOpen),
    NuFile::writeMode,
    FALSE // do not open the file.
);

// Get the file descriptor of the file.
INT file_descriptor = file->GetDescriptor();
```

### class NuDirectory

NuDirectory objects encapsulate a directory on a storage device.

#### Public Member Functions

Member	Overview
NuDirectory	Constructor
~NuDirectory	Destructor
Make	Creates the physical directory on the disk.
Remove	Removes the physical directory on the disk.
GetFirst	Used to start an iteration session for traversing the directory entries within the actual physical directory on the disk that the directory object encapsulates.
GetNext	Used to retrieve the next entry in an iteration session.
Done	Used to complete an iteration session.
GetPath	Returns the path specification within the object.
SetPath	Changes the path specification within the object. This does not rename the directory on the physical disk.



**NuDirectory::NuDirectory**`NuDirectory();`

Default constructor

**Overview**

Condition	Description
Pre-condition	None
Action	Constructs a NuDirectory object.
Post-condition	The path information within the object is empty ("").

**Parameters**

None

**Return Value**

None

**Example**

```
// Construct a directory object.
NuDirectory* directory = new NuDirectory;

// Set the path.
directory->SetPath( "my_directory" );
```



### NuDirectory::NuDirectory

```
NuDirectory( const CHAR* init_path );
```

Constructor that takes an initial path.

#### Overview

Condition	Description
Pre-condition	None
Action	Constructs a NuDirectory object.
Post-condition	The directory object exists (not necessarily the physical directory on the disk, just the object). The path is set to the supplied path parameter.

#### Parameters

Parameter	Overview
init_path	The initial path name.

#### Return Value

None

#### Example

```
// Create a directory object for my directory.  
NuDirectory* directory = new NuDirectory( "my_directory" );
```



**NuDirectory::~~NuDirectory**

```
virtual  
~NuDirectory();
```

Destructor

**Overview**

Condition	Description
Pre-condition	None
Action	Returns all resources to the system.
Post-condition	The object does not exist.

**Parameters**

None

**Return Value**

None

**Example**

Destructors are called automatically when the object goes out of scope or is deleted.

### NuDirectory::Make

```
virtual  
STATUS  
Make() ;
```

Creates the physical directory on the disk.

#### Overview

Condition	Description
Pre-condition	The directory does not exist on the disk.
Action	Creates a physical directory on the disk.
Post-condition	The physical directory exists.

#### Parameters

None

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS if successful, otherwise returns one of the following error codes:  NUF_BAD_USER – Not a file user. NUF_BADDRIVE – Invalid drive specified. NUF_NOSPC – No space to create directory NUF_LONGPATH – Path or directory name too long. NUF_INVNAME – Path or filename includes invalid character. NUF_ROOT_FULL – Root directory full. NUF_NOFILE – The specified file not found. NUF_EXIST – The directory already exists. NUF_NO_BLOCK – No block buffer available. NUF_NO_FINODE – No FINODE buffer available. NUF_NO_DROBJ – No DROBJ buffer available. NUF_IO_ERROR – IO error occurred. NUF_INTERNAL – Nucleus FILE internal error.

#### Example

```
// Create a directory object for my directory.  
NuDirectory* directory = new NuDirectory( "my_directory" );  
  
// Physically make the directory on the disk.  
if( NU_SUCCESS == directory->Make() )  
{  
    // The directory was successfully created.  
}
```





**NuDirectory::Remove**

```
virtual
STATUS
Remove();
```

Removes the physical directory on the disk.

**Overview**

Condition	Description
Pre-condition	The directory exists on the disk.
Action	Removes a physical directory on the disk.
Post-condition	The physical directory does not exist.

**Parameters**

None

**Return Value**

Return Value	Overview
STATUS	<p>Returns NU_SUCCESS if successful, otherwise returns one of the following error codes:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_BADPARM – Invalid parameter specified.</p> <p>NUF_INVNAME – Path or filename includes invalid character.</p> <p>NUF_NOFILE – The specified file not found.</p> <p>NUF_ACCES – This file has at least one of the following attributes: RDONLY, HIDDEN, SYSTEM, VOLUME.</p> <p>NUF_SHARE – The access conflict from multiple tasks to a specific file.</p> <p>NUF_NO_FINODE – No FINODE buffer available.</p> <p>NUF_NO_DROBJ – No DROBJ buffer available.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

**Example**

```
// Create a directory object for my directory.
NuDirectory* directory = new NuDirectory( "my_directory" );

// Physically remove the directory on the disk.
if( NU_SUCCESS == directory->Remove() )
{
    // The directory was successfully removed.
}
```



## NuDirectory::GetFirst

```
virtual
STATUS
GetFirst( DSTAT* statobj );
```

Used to start an iteration session for traversing the directory entries within the actual physical directory on the disk that the directory object encapsulates.

### Overview

Condition	Description
Pre-condition	None
Action	Sets up the iteration session.
Post-condition	The iteration session is in progress.

### Parameters

Parameter	Overview
statobj	A DSTAT structure that identifies the instance of the directory iteration session. The same structure is used upon subsequent calls to the GetNext and Done members of the object.

### Return Value

Return Value	Overview
STATUS	<p>NU_SUCCESS is returned upon success, otherwise one of the following error codes is returned:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_INVNAME – Path or filename includes invalid character.</p> <p>NUF_NOFILE – The specified file not found.</p> <p>NUF_NO_FINODE – No FINODE buffer available.</p> <p>NUF_NO_DROBJ – No DROBJ buffer available.</p> <p>NUF_IO_ERROR – Driver IO function routine returned error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>



**Example**

```

// Go through the directory and insert copies of DSTAT structures in a
list
DSTAT statobj;

// Start the search.
if( directory.GetFirst( &statobj ) )
{
    list<DSTAT> testList;

    do
    {
        // Insert it onto the back of the list.
        testList.push_back( statobj );

    } while( directory.GetNext( &statobj ) );

    // Close the search.
    directory.Done( &statobj );

    // Print the filename of all files in the directory.
    list<DSTAT>::iterator testListIterator = testList.begin();
    while( testListIterator != testList.end() )
    {
        cout << (*testListIterator++).fname; // the filename.
    }

    // Remove the elements from the list.
    testList.erase( testList.begin(), testList.end() );
}

```

### NuDirectory::GetNext

virtual

STATUS

GetNext( DSTAT\* statobj );

Used to retrieve the next entry in an iteration session.

#### Overview

Condition	Description
Pre-condition	The <code>GetFirst</code> member has been called on the object.
Action	Fills in the <code>DSTAT</code> structure with information for the next file in the iteration session.
Post-condition	The <code>DSTAT</code> structure contains information for the next file in the iteration session.

#### Parameters

Parameter	Overview
statobj	A <code>DSTAT</code> structure that identifies the instance of the directory iteration session. The same structure is used upon subsequent calls to the <code>GetNext</code> and <code>Done</code> members of the object.

#### Return Value

Return Value	Overview
STATUS	<code>NU_SUCCESS</code> is returned upon success, otherwise one of the following error codes is returned:  <code>NUF_BAD_USER</code> – Not a file user. <code>NUF_BADPARM</code> – Invalid parameter specified. <code>NUF_NOFILE</code> – The specified file not found. <code>NUF_NO_FINODE</code> – No <code>FINODE</code> buffer available. <code>NUF_IO_ERROR</code> – Driver IO function routine returned error. <code>NUF_INTERNAL</code> – Nucleus FILE internal error.



**Example**

```

// Go through the directory and insert copies of DSTAT structures in a
list
DSTAT statobj;

// Start the search.
if( directory.GetFirst( &statobj ) )
{
    list<DSTAT> testList;

    do
    {
        // Insert it onto the back of the list.
        testList.push_back( statobj );

    } while( directory.GetNext( &statobj ) );

    // Close the search.
    directory.Done( &statobj );

    // Print the filename of all files in the directory.
    list<DSTAT>::iterator testListIterator = testList.begin();
    while( testListIterator != testList.end() )
    {
        cout << (*testListIterator++).fname; // the filename.
    }

    // Remove the elements from the list.
    testList.erase( testList.begin(), testList.end() );
}

```

### NuDirectory::Done

```
virtual  
STATUS  
Done( DSTAT* statobj );
```

Used to complete an iteration session.

#### Overview

Condition	Description
Pre-condition	The <code>GetFirst</code> member has been called on the object.
Action	Completes the iteration session.
Post-condition	The iteration session is complete.

#### Parameters

Parameter	Overview
<code>statobj</code>	A <code>DSTAT</code> structure that identifies the instance of the directory iteration session.

#### Return Value

Return Value	Overview
<code>STATUS</code>	<code>NU_SUCCESS</code> is returned.

#### Example

```
// Go through the directory and insert copies of DSTAT structures  
in a list DSTAT statobj;  
  
// Start the search.  
if( directory.GetFirst( &statobj ) )  
{  
    list<DSTAT> testList;  
  
    do  
    {  
        // Insert it onto the back of the list.  
        testList.push_back( statobj );  
    } while( directory.GetNext( &statobj ) );  
  
    // Close the search.  
    directory.Done( &statobj );  
  
    // Print the filename of all files in the directory.  
    list<DSTAT>::iterator testListIterator = testList.begin();  
    while( testListIterator != testList.end() )  
    {  
        cout << (*testListIterator++).fname; // the filename.  
    }  
  
    // Remove the elements from the list.  
    testList.erase( testList.begin(), testList.end() );  
}
```



**NuDirectory::GetPath**

```
const
CHAR*
GetPath() const;
```

Returns the path specification within the object.

**Overview**

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
CHAR*	A valid <code>const</code> pointer to the path specification within the directory object.

**Example**

```
// Construct a directory object.
NuDirectory* directory = new NuDirectory( "my_directory" );

// Get the path.
const CHAR* path = directory->GetPath();
```

### NuDirectory::SetPath

```
void  
SetPath( const CHAR* source_path );
```

Changes the path specification within the object. This does not rename the directory on the physical disk.

#### Overview

Condition	Description
Pre-condition	None
Action	Copies the string into the object.
Post-condition	The object's path is set.

#### Parameters

Parameter	Overview
source_path	A const pointer to the path source.

#### Return Value

None

#### Example

```
// Construct a directory object.  
NuDirectory* directory = new NuDirectory;  
  
// Set the path.  
directory->SetPath( "my_directory" );
```





```
class NppFile : public NppComponent
```

The Nucleus C++ FILE component class.

### Public Member Functions

Member	Overview
NppFile	Constructor
Initialize	Initializes the underlying Nucleus FILE system.
~NppFile	Destructor
BecomeUser	Registers the calling thread as a user of the file system.
ReleaseUser	Un-registers the calling thread as a user of the file system.
GetNumberUsers	Returns the number of simultaneous threads the file system is setup to support.
GetNumberUserFiles	Returns the number of files each user of the file system can have open at any one time.
GetNumberDevices	Returns the number of storage devices setup in the file system.
GetMaxSectors	Returns the maximum number of sectors that will be read or written in any one internal transaction.
GetLockMethod	Returns the current Nucleus FILE lock method.

### NppFile::NppFile

NppFile();

Constructor

#### Overview

Condition	Description
Pre-condition	None
Action	Creates the object.
Post-condition	The object exists, but the file system is not initialized. The <code>Initialize</code> member is used for that.

#### Parameters

None

#### Return Value

None

#### Example

```
// Nucleus C++ FILE component.  
NppFILE* nppFILE = new NppFILE;  
nppFILE->Initialize();  
  
// Nucleus C++ FILE device.  
NuFileDevice* nppFILE_Device = new NuFileDevice( 0 /*A:*/ );  
nppFILE_Device->Initialize();
```



**NppFile::Initialize**

```
virtual
STATUS
Initialize();
```

Initializes the underlying Nucleus FILE system.

**Overview**

Condition	Description
Pre-condition	None
Action	Calls Nucleus FILE initialization code for the file system.
Post-condition	Nucleus FILE is initialized and ready to use.

**Parameters**

None

**Return Value**

Return Value	Overview
STATUS	Returns the result initializing Nucleus FILE. This is specific to your version of Nucleus FILE. Please refer to Nucleus FILE documentation for a list of possible return values for your target.

**Example**

```
// Nucleus C++ FILE component.
NppFILE* nppFILE = new NppFILE;
nppFILE->Initialize();

// Nucleus C++ FILE device.
NuFileDevice* nppFILE_Device = new NuFileDevice( 0 /*A:*/ );
nppFILE_Device->Initialize();
```

### NppFile::~NppFile

Virtual  
~NppFile();

Destructor

#### Overview

Condition	Description
Pre-condition	None
Action	Returns all resources to the system. <b>NOTE:</b> Nucleus FILE resources are not returned.
Post-condition	The component does not exist.

#### Parameters

None

#### Return Value

None

#### Example

Destructors are called automatically when the object goes out of scope or is deleted.



**NppFile::BecomeUser**

```
virtual
INT
BecomeUser();
```

Registers the calling thread as a user of the file system.

**Overview**

Condition	Description
Pre-condition	The thread is not registered as a file system user.
Action	Registers the thread as a file system user.
Post-condition	The thread is registered as a file system user.

**Parameters**

None

**Return Value**

Return Value	Overview
INT	Returns YES if the task may use the file system or NO if too many users are already using it.

**Example**

```
// Register with the file system.
nppFILE->BecomeUser();

// Use the file system, storage devices, sub-directories, files, etc.
// ...

// Un-Register with the file system.
nppFILE->ReleaseUser();
```

### NppFile::ReleaseUser

```
virtual  
VOID  
ReleaseUser();
```

Un-registers the calling thread as a user of the file system.

#### Overview

Condition	Description
Pre-condition	The thread has been previously registered using the <code>BecomeUser</code> member.
Action	Releases the file system resources for the calling thread.
Post-condition	The calling thread is not a registered user of the file system.

#### Parameters

None

#### Return Value

None

#### Example

```
// Register with the file system.  
nppFILE->BecomeUser();  
  
// Use the file system, storage devices, sub-directories, files,  
// etc.  
// ...  
  
// Un-Register with the file system.  
nppFILE->ReleaseUser();
```



**NppFile::GetNumberUsers**

UINT16

GetNumberUsers() const;

Returns the number of simultaneous threads the file system is setup to support.

**Overview**

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
UINT16	Returns the number of simultaneous threads the file system is setup to support.

**Example**

```
// Get a pointer to the file system.
NppFILE* nppFILE = NppFILE::Instance();

// Get the number of users.
UINT16 number_users = NppFILE->GetNumberUsers();
```

### NppFile::GetNumberUserFiles

UINT16

GetNumberUserFiles() const;

Returns the number of files each user of the file system can have open at any one time.

#### Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

#### Parameters

None

#### Return Value

Return Value	Overview
UINT16	Returns the number of files each user of the file system can have open at any one time.

#### Example

```
// Get a pointer to the file system.
NppFILE* nppFILE = NppFILE::Instance();

// Get the number of user files.
UINT16 number_user_files = NppFILE->GetNumberUserFiles();
```





**NppFile::GetNumberDevices**

UINT16

GetNumberDevices() const;

Returns the number of storage devices setup in the file system.

**Overview**

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
UINT16	Returns the number of storage devices setup in the file system.

**Example**

```
// Get a pointer to the file system.
NppFILE* nppFILE = NppFILE::Instance();

// Get the number of devices.
UINT16 number_devices = NppFILE->GetNumberDevices();
```

### NppFile::GetMaxSectors

UINT16

GetMaxSectors() const;

Returns the maximum number of sectors that will be read or written in any one internal transaction.

#### Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

#### Parameters

None

#### Return Value

Return Value	Overview
UINT16	Returns the maximum number of sectors that will be read or written in any one internal transaction.

#### Example

```
// Get a pointer to the file system.
NppFILE* nppFILE = NppFILE::Instance();

// Get the max sectors.
UINT16 number_max_sectors = NppFILE->GetMaxSectors();
```



**NppFile::GetLockMethod**

UINT16

GetLockMethod() const;

Returns the current Nucleus FILE lock method.

**Overview**

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
UINT16	Returns the current Nucleus FILE lock method.

**Example**

```
// Get a pointer to the file system.
NppFILE* nppFILE = NppFILE::Instance();

// Get the lock method.
UINT16 lock_method = NppFILE->GetLockMethod();
```

## class NuFileDevice

This is an encapsulation of an installed Nucleus FILE driver. Nucleus C++ FILE uses the drivers you setup when you build the Nucleus FILE library. Please refer to Nucleus FILE documentation for how to setup a Nucleus FILE device driver. The `NuFileDevice` class provides a class interface for these storage devices.

### Public Member Functions

Member	Overview
<code>NuFileDevice();</code>	Constructor
<code>Initialize();</code>	Initializes the device.
<code>~NuFileDevice();</code>	Destructor
<code>SetFormatParameters();</code>	Sets up the format parameters stored within the object. This does not format the device.
<code>GetFormatParameters();</code>	Returns a <code>const</code> pointer to this object's format parameters structure.
<code>Format();</code>	Format the device based on the current format parameters setup within the object.
<code>Open();</code>	Opens the device for use by the calling thread.
<code>Close();</code>	Closes the device for the calling thread.
<code>Abort();</code>	Frees all resources associated with this storage device and invalidates all file descriptors. This method does not perform any disk writes.
<code>SetDefaultDevice();</code>	Sets this storage device as the default device for the calling thread.
<code>GetDefaultDevice();</code>	Returns a pointer to the storage device that is the default device for the calling thread.
<code>GetCurrentDirectory();</code>	Fills in the destination path with the current working directory for the calling thread on this storage device.
<code>SetCurrentDirectory();</code>	Sets the current working directory for the calling thread for this storage device.
<code>GetDeviceNumber();</code>	Returns the device number associated with this storage device.

### Public Class Member Functions

Member	Overview
<code>GetDevice();</code>	Returns a pointer to the storage device associated with the supplied device number.

### Protected Member Functions

Member	Overview
<code>PrependDeviceNameToName();</code>	A utility routine that adds the device letter to the supplied path, if one does not already exist.



**NuFileDevice::NuFileDevice**

```
NuFileDevice( UINT16 init_device_number );
```

Constructor

**Overview**

Condition	Description
Pre-condition	The Nucleus C++ FILE component has been initialized.
Action	None
Post-condition	The object exists.

**Parameters**

Parameter	Overview
init_device_number	The device number associated with this storage device object.

**Return Value**

None

**Example**

```
// Nucleus C++ FILE component.
NppFILE* nppFILE = new NppFILE;
nppFILE->Initialize();

// Nucleus C++ FILE device.
NuFileDevice* nppFILE_Device = new NuFileDevice( 0 /*A:*/ );
nppFILE_Device->Initialize();
```

### NuFileDevice::Initialize

Virtual  
STATUS  
Initialize();

Initializes the device.

#### Overview

Condition	Description
Pre-condition	None
Action	Registers this object with the class as the storage device associated with the device number stored in the object.
Post-condition	The object is initialized and ready to use.

#### Parameters

None

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS.

#### Example

```
// Nucleus C++ FILE component.  
NppFILE* nppFILE = new NppFILE;  
nppFILE->Initialize();  
  
// Nucleus C++ FILE device.  
NuFileDevice* nppFILE_Device = new NuFileDevice( 0 /*A:*/ );  
nppFILE_Device->Initialize();
```



**NuFileDevice::~~NuFileDevice**

```
virtual  
~NuFileDevice();
```

Destructor

**Overview**

Condition	Description
Pre-condition	None
Action	Destructs the object. All resources are returned to the system.
Post-condition	The object does not exist.

**Parameters**

None

**Return Value**

None

**Example**

Destructors are called automatically when the object goes out of scope or is deleted.

### NuFileDevice::SetFormatParameters

```
virtual  
STATUS  
SetFormatParameters( const FMTPARMS* format_parameters );
```

Sets up the format parameters stored within the object. This does not format the device.

#### Overview

Condition	Description
Pre-condition	The object has been initialized.
Action	Copies the supplied structure into the object but does not format the device.
Post-condition	The object's format parameters contain the supplied parameters.

#### Parameters

Parameter	Overview
format_parameters	A pointer to a format parameter structure to copy.

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS.

#### Example

```
// Get the current parameters from the device.  
const FMTPARMS* parameters = device->GetFormatParameters();  
  
// Format the device using the same format parameters.  
device->SetFormatParameters( parameters );  
device->Format();
```





**NuFileDevice::GetFormatParameters**

```
virtual
const
FMTPARMS*
GetFormatParameters() const;
```

Returns a `const` pointer to this object's format parameters structure.

**Overview**

Condition	Description
Pre-condition	None
Action	Returns a <code>const</code> pointer to the internal structure.
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
STATUS	Returns a <code>const</code> pointer to the internal structure.

**Example**

```
// Get the current parameters from the device.
const FMTPARMS* parameters = device->GetFormatParameters();
```

## NuFileDevice::Format

```
virtual
STATUS
Format();
```

Format the device based on the current format parameters setup within the object.

### Overview

Condition	Description
Pre-condition	None
Action	Formats the device.
Post-condition	The device is formatted.

### Parameters

None

### Return Value

Return Value	Overview
STATUS	<p>Returns NU_SUCCESS upon successful completion of the service, otherwise one of the following error codes is returned:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_NOT_OPENED – The disk is not opened yet.</p> <p>NUF_FATCORE – Fat cache table too small.</p> <p>NUF_BADPARAM – Invalid parameter specified.</p> <p>NUF_BADDISK – Bad Disk.</p> <p>NUF_NO_PARTITION – No partition in disk.</p> <p>NUF_NOFAT – No FAT type in this partition.</p> <p>NUF_FMTCSIZE – Too many clusters for this partition.</p> <p>NUF_FMTFSIZE – File allocation table too small.</p> <p>NUF_FMTRSIZE – Numroot must be an even multiple of 16.</p> <p>NUF_INVNAME – Volume label includes invalid character.</p> <p>NUF_NO_MEMORY – Can't allocate internal buffer.</p> <p>NUF_NO_BLOCK – No block buffer available.</p> <p>NUF_IO_ERROR – Driver IO error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>



**Example**

```
// Get the current parameters from the device.  
const FMTPARMS* parameters = device->GetFormatParameters();  
  
// Format the device using the same format parameters.  
device->SetFormatParameters( parameters );  
device->Format();
```

### NuFileDevice::Open

```
virtual  
STATUS  
Open();
```

Opens the device for use by the calling thread.

#### Overview

Condition	Description
Pre-condition	None
Action	Opens the device for the calling thread.
Post-condition	The device is open for the calling thread.

#### Parameters

None

#### Return Value

Return Value	Overview
STATUS	Returns NU_SUCCESS upon successful completion of the service, otherwise one of the following error codes is returned:  NUF_BAD_USER – Not a file user. NUF_BADDRIIVE – Invalid drive specified. NUF_FATCORE – Fat cache table too small. NUF_NO_PARTITION – No partition in disk. NUF_FORMAT – Not formatted this disk. NUF_NO_MEMORY – Can't allocate internal buffer. NUF_IO_ERROR – Driver returned error. NUF_INTERNAL – Nucleus FILE internal error.

#### Example

```
// Open the storage device.  
device->Open();  
  
// Access the storage device, create files, sub-directories, etc.  
// ...  
  
// Close the storage device to return resources.  
device->Close();
```



**NuFileDevice::Close**

```
virtual
INT
Close();
```

Closes the device for the calling thread.

**Overview**

Condition	Description
Pre-condition	The device is open for the calling thread.
Action	Closes the device for the calling thread.
Post-condition	The device is closed for the calling thread.

**Parameters**

None

**Return Value**

Return Value	Overview
INT	<p>Returns NU_SUCCESS upon successful completion of the service, otherwise one of the following error codes is returned:</p> <p>NUF_BAD_USER – Not a file user.  NUF_BADDRIVE – Invalid drive specified.  NUF_NOT_OPENED – Drive not opened.  NUF_IO_ERROR – IO_error occurred.  NUF_INTERNAL – Nucleus FILE internal error.</p>

**Example**

```
// Open the storage device.
device->Open();

// Access the storage device, create files, sub-directories, etc.
// ...

// Close the storage device to return resources.
device->Close();
```

### NuFileDevice::Abort

```
virtual  
VOID  
Abort();
```

Frees all resources associated with this storage device and invalidates all file descriptors. This method does not perform any disk writes.

#### Overview

Condition	Description
Pre-condition	None
Action	Frees all resources associated with this storage device. This method does not perform any disk writes.
Post-condition	All file descriptors are invalid.

#### Parameters

None

#### Return Value

None

#### Example

```
// Abort all operations.  
device->Abort();  
  
// Go ahead and start over.  
device->Open();  
  
// etc.  
// ...
```



**NuFileDevice::SetDefaultDevice**

```
virtual
STATUS
SetDefaultDevice();
```

Sets this storage device as the default device for the calling thread.

**Overview**

Condition	Description
Pre-condition	None
Action	Sets this storage device as the default device for the calling thread.
Post-condition	This storage device is the default device for the calling thread.

**Parameters**

None

**Return Value**

Return Value	Overview
STATUS	Returns NU_SUCCESS upon successful completion of the service, otherwise one of the following error codes is returned:  NUF_BAD_USER – Not a file user. NUF_INVNAME – If the drive is out of range.

**Example**

```
// Make this the default device.
device->SetDefaultDevice();
```

### NuFileDevice::GetDefaultDevice

```
static  
NuFileDevice*  
GetDefaultDevice();
```

Returns a pointer to the storage device that is the default device for the calling thread.

#### Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

#### Parameters

None

#### Return Value

Return Value	Overview
NuFileDevice*	Returns a pointer to the storage device that is the default device for the calling thread. If there is not a default device, NULL is returned.

#### Example

```
// Get the default device. Note: this is a static class member.  
NuFileDevice* device = NuFileDevice::GetDefaultDevice();
```





**NuFileDevice::GetCurrentDirectory**

```
virtual
STATUS
GetCurrentDirectory( CHAR* pathDestination ) const;
```

Fills in the destination path with the current working directory for the calling thread on this storage device.

**Overview**

Condition	Description
Pre-condition	None
Action	Fills in the destination path with the current working directory for the calling thread on this storage device.
Post-condition	None

**Parameters**

Parameter	Overview
pathDestination	A destination buffer that receives the current working directory (EMAXPATH is the max length copied).

**Return Value**

Return Value	Overview
STATUS	Returns NU_SUCCESS upon successful completion of the service, otherwise one of the following error codes is returned:  NUF_BAD_USER – Not a file user. NUF_INVNAME – Disk not opened yet.

**Example**

```
// Retrieve the current working directory for this thread.
CHAR path[EMAXPATH];
device->GetCurrentDirectory( path );
```

## NuFileDevice::SetCurrentDirectory

```
virtual
STATUS
SetCurrentDirectory( const CHAR* name);
```

Sets the current working directory for the calling thread for this storage device.

### Overview

Condition	Description
Pre-condition	None
Action	Sets the current working directory for the calling thread for this storage device.
Post-condition	The current working directory for the calling thread for this storage device is set.

### Parameters

Parameter	Overview
Name	A pointer to string that holds the new current working directory.

### Return Value

Return Value	Overview
STATUS	<p>Returns NU_SUCCESS upon successful completion of the service, otherwise one of the following error codes is returned:</p> <p>NUF_BAD_USER – Not a file user.</p> <p>NUF_BADDRIVE – Invalid drive specified.</p> <p>NUF_LONGPATH – Path or directory name too long.</p> <p>NUF_ACCES – Not a directory attributes.</p> <p>NUF_NOFILE – The specified file not found.</p> <p>NUF_NO_BLOCK – No block buffer available.</p> <p>NUF_NO_FINODE – No FINODE buffer available.</p> <p>NUF_NO_DROBJ – No DROBJ buffer available.</p> <p>NUF_IO_ERROR – Driver IO function routine returned error.</p> <p>NUF_INTERNAL – Nucleus FILE internal error.</p>

### Example

```
// Change the current working directory.
device->SetCurrentDirectory( "\\Another_Working_Directory" );
```



**NuFileDevice::GetDeviceNumber**

UINT16

GetDeviceNumber();

Returns the device number associated with this storage device.

**Overview**

Condition	Description
Pre-condition	None
Action	Returns the device number associated with this storage device.
Post-condition	None

**Parameters**

None

**Return Value**

Return Value	Overview
UINT16	The device number associated with this storage device.

**Example**

```
UINT16 device_number = device->GetDeviceNumber();
```

### NuFileDevice::GetDevice

```
static  
NuFileDevice*  
GetDevice( UINT16 device_number );
```

Returns a pointer to the storage device associated with the supplied device number.

#### Overview

Condition	Description
Pre-condition	None
Action	Returns a pointer to the storage device associated with the supplied device number.
Post-condition	None

#### Parameters

Parameter	Overview
device_number	The device number associated with the requested NuFileDevice.

#### Return Value

Return Value	Overview
NuFileDevice*	A pointer to the storage device associated with the supplied device number.

#### Example

```
// Get a pointer to the "A:" device.  
NuFileDevice* device = NuFileDevice::GetDevice( 0 /*"A:"*/ );
```



**NuFileDevice::PrependDeviceNameToName**

virtual

VOID

PrependDeviceNameToName( CHAR\* target, const CHAR\* name ) const;

A utility routine that adds the device letter to the supplied path, if one does not already exist.

**Overview**

Condition	Description
Pre-condition	None
Action	Adds the device letter to the supplied path, if one does not already exist.
Post-condition	The device letter is added to the supplied path.

**Parameters**

Parameter	Overview
target	A pointer to the buffer that is to be modified to include altered path with the addition of the drive letter.
name	The buffer that holds the original path information.

**Return Value**

None

**Example**

```
CHAR* path = "\\My_path";

CHAR absolute_path[EMAXPATH];
PrependDeviceNameToName( absolute_path, path );

// absolute_path now contains "A:\MyPath"
```

